# 用户行为分析（扩展）

01

# 实时用户检测

## 需求分析

3秒之内连续失败2次，则进行风控

## 数据展示

| 用户ID | IP | 用户行为 | 时间 |
| --- | --- | --- | --- |
| 5692 | 66.249.3.15 | fail | 1558430844 |


1. 设置state，存储失败任务
2. 3s之内超过2次，告警

# 实现思路

1. 乱序的数据不好处理
2. 很难应对复杂多变风控的需求

## 1、什么是CEP

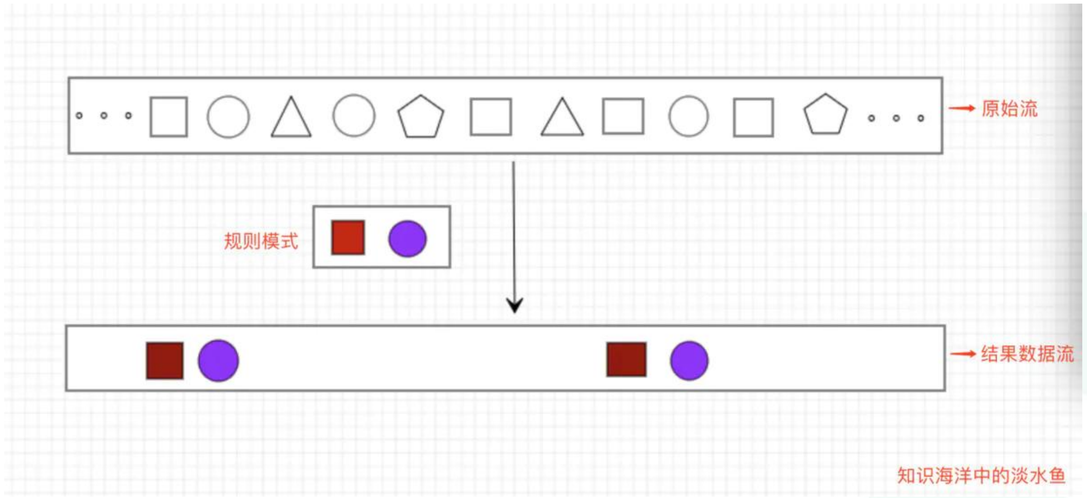
复杂事件处理(Complex Event Processing, CEP)

## 2、什么是复杂事件

复杂事件：检测和发现无界事件流中多个记录的关联规则

CEP(Complex Event Processing)就是在无界事件流中检测事件模式，让我们掌握数据中重要的部分。

Flink CEP是在flink中实现的复杂事件处理库。



## 1.读取数据

```
al loginEventStream = env.addSource(.....)
```

## 2.定义匹配规则

```
val loginFailPattern = Pattern.begin[LoginEvent]( name = "begin").where(_.eventType == "fail")
  .next( name = "next").where(_.eventType == "fail")
  .within(Time.seconds( seconds = 3))
```

## 3. 在事件流上应用匹配规则

```
val patternStream = CEP.pattern(loginEventStream, loginFailPattern)
```

## 4. 提取匹配事件

```
val loginFailDataStream = patternStream.select( new LoginFailMatch() )
```

- **个体模式(Individual Patterns)**

  -组成复杂规则的每一个单独的模式定义，就是“个体模式”

  start.times (3).where(_.behaviar.startsWith(“test”))

- **序列模式(Combining Patterns)**

  -很多个体模式组合起来，就形成了整个的模式序列

  -模式序列必须以一个“初始模式”开始：

  val start = Pattern.begin("start")

- **模式组(Groups of patterns)**

  -将一个序列模式作为条件嵌套在个体模式里，成为一组模式

## 个体模式

包括“单例(singleton)模式”和“循环(looping)模式” 单例模式只接收一个事件，而循环模式可以接收多个，通过量词(Quantifier)指定。

## 量词

可以在一个个体模式后追加量词，也就是指定循环次数

```
// expecting 4 occurrences
start.times(4)

// expecting 0 or 4 occurrences
start.times(4).optional()

// expecting 2, 3 or 4 occurrences
start.times(2, 4)
```

```
// expecting 1 or more occurrences
start.oneOrMore()

// expecting 0, 2 or more occurrences and repeating as many as possible
start.timesOrMore(2).optional().greedy()
```

（1）times：指定固定的循环执行次数
（2）optional：通过Optional关键字指定要么不触发要么触发指定的次数
（3）greedy：贪婪模式，在pattern匹配成功的前提下会尽可能多的触发
（4）oneOrMore：指定触发一次或多次
（5）timesOrMore：指定触发固定次数以上

# 条件

每个模式都需要指定触发条件，作为模式是否接受事件进入的判断依据，CEP中的个体模式主要通过调用.where() .or()和.until。来指定条件按不同的调用方式，可以分成以下几类：简单条件，组合条件，迭代条件，终止条件

## 简单条件(Simple Condition)

通过.where()方法对事件中的字段进行判断筛选，决定是否接受该事件

```
start.where(event => event.getName.startsWith("foo"))
```

## 组合条件(Combining Condition)

将简单条件进行合并；.or()方法表示或逻辑相连，where的直接组合就是AND

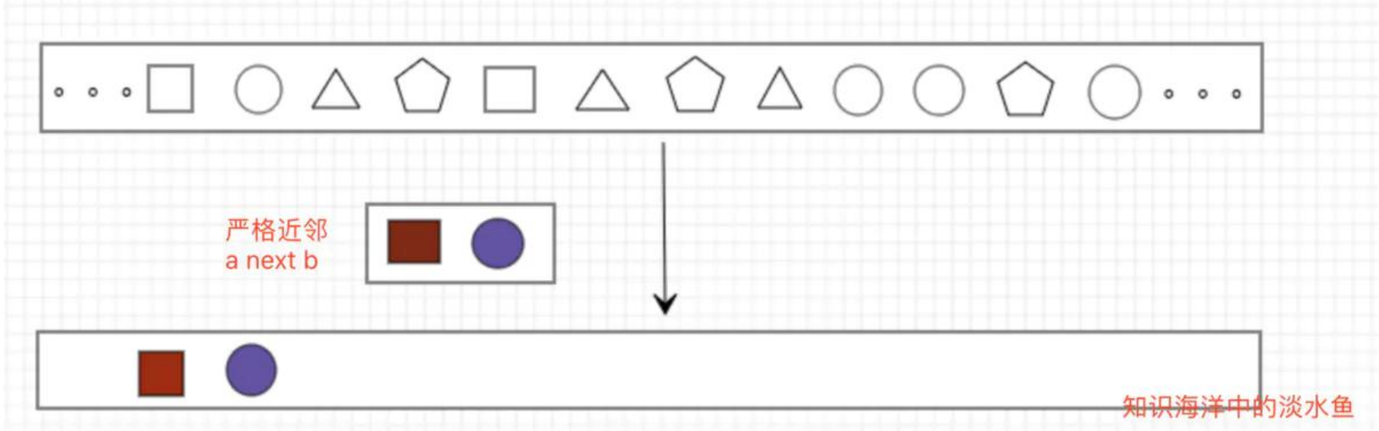
```
pattern.where(event => ... /* some condition */).or(event => ... /* or condition */)
```

## 终止条件(Stop Condition)

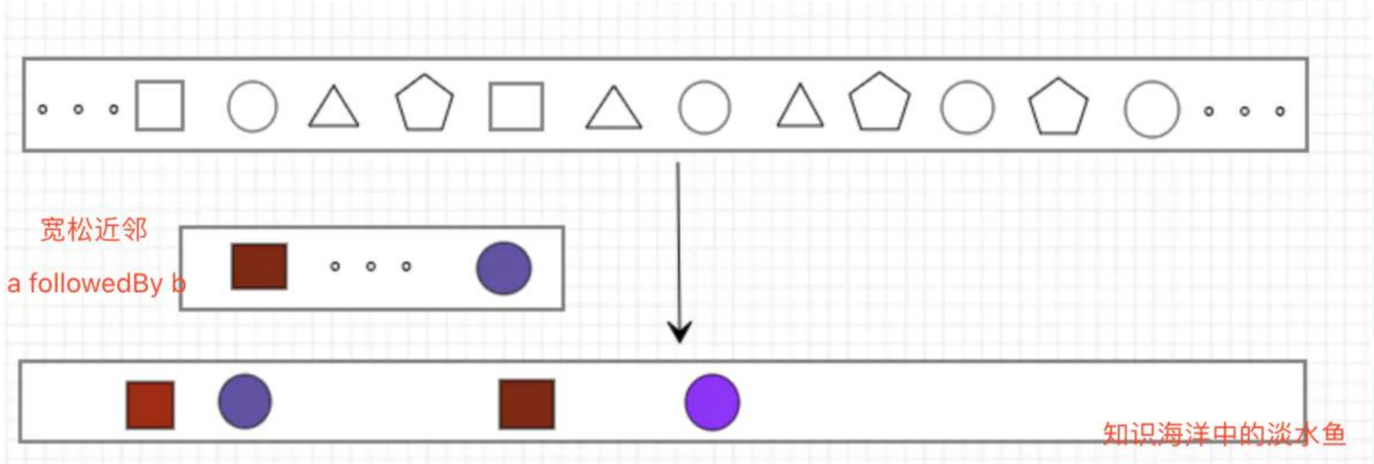
如果使用了 oneOrMore或者oneOrMore.optional

建议使用.until()作为终止条件

严格近邻




宽松近邻



## 严格近邻(Strict Contiguity)

所有事件按照严格的顺序出现，中间没有任何不匹配的事件，由.next。指定

例如对于模式” a next b” ,事件序列［a, c, bl, b2］没有匹配

## 宽松近邻(Relaxed Contiguity)

允许中间出现不匹配的事件，由.followedBy()指定

例如对于模式” a followed By b",事件序列［a,c,bl,b2］匹配为{a, bl}

## 非确定性宽松近邻(Non-Deterministic Relaxed Contiguity)

进一步放宽条件，之前已经匹配过的事件也可以再次使用，由.followedByAny()指定

例如对于模式” a followedByAny b” ,事件序列［a, c, bl, b2］匹配为{a, bl), {a, b2)

## 注意

1. 如果不希望出现某种近邻关系：

   notNext()——不想让某个事件严格紧邻前一个事件发生

   notFollowedBy()——不想让某个事件在两个事件之间发生

2. 所有模式序列必须以.begin开始

3. 模式序列不能以.notFollowedBy()结束

4. “not”类型的模式不能被0ptional所修饰

5. 可以为模式指定时间约束，用来要求在多长时间内匹配有效

```
val loginFailPattern = Pattern.begin[LoginEvent]( name = "begin").where(_.eventType == "fail")
  .next( name = "next").where(_.eventType == "fail")
  .within(Time.seconds( seconds = 3))
```

指定要查找的模式序列后，就可以将其应用于输入流以检测潜在匹配

```
val patternStream = CEP.pattern(loginEventStream, loginFailPattern)
```

```
val loginFailDataStream = patternStream.select( new LoginFailMatch() )
```

```
class LoginFailMatch() extends PatternSelectFunction[LoginEvent, Warning]{
  override def select(map: util.Map[String, util.List[LoginEvent]]): Warning = {
    // 从map中按照名称取出对应的事件
    val firstFail = map.get("begin").iterator().next()
    val lastFail = map.get("next").iterator().next()
    Warning( firstFail.userId, firstFail.eventTime, lastFail.eventTime, "login fail!" )
  }
}
```

## 需求分析

三分钟之内连续出现三次及以上的温度高于40度，就是异常温度，进行告警

## 数据展示

| 传感器设备地址 | 时间 | 温度 |
| --- | --- | --- |
| 01-34-5E-5F-33-W2 | 2020-12-23 11:10:10 | 35 |

奈学教育，一个有干货更有温度的教育品牌

出品： 奈学教育